Clase 2 — 14.10.25

#IntelliJIDEA #JAVA

📘 Acceso a Datos

🗓️ Clase 2— 14/10/2025

🎯 Tema: Creación, gestión y eliminación de directorios y ficheros en Java

---

🧩 Propuesta del Ejercicio 1

🔹 Conceptos clave

- Importar las clases necesarias:

```
import java.io.File;
import java.io.IOException;
```

- **File**: clase que representa archivos o directorios.
- **IOException**: excepción que puede lanzar `createNewFile()`.

🔹 Flujo general

1. **Declaración de rutas**

```
    File miDirectorio = new File("C:\\AD\\Ejercicio1\\miDirectorio");
    File miFichero = new File("C:\\AD\\Ejercicio1\\miDirectorio\\miFichero.txt");
```

- Diferenciar **“apuntar” a la ruta** (crear el objeto `File`) de **crear físicamente** el directorio o fichero.

2. **Creación de directorio**

```
miDirectorio.mkdirs();
```

- `mkdir()` crea **solo un nivel**.
- `mkdirs()` crea **toda la ruta completa** si no existe.

3. **Creación del fichero**

```
    if (miFichero.createNewFile()) {
        System.out.println("Fichero creado         correctamente");
    } else {
        System.out.println("ERROR: No se           ha podido crear el fichero");
    }
```

- `createNewFile()` devuelve **true** si lo crea y **false** si ya existe.

1. **Manejo de excepciones**

```
    try {
        ...
    } catch (IOException e) {
        System.out.println("Error al crear         el fichero: " + e.getMessage());
    }
```

```
- Captura errores de entrada/salida.

- Alternativa:
```

```
e.printStackTrace();
```

Muestra la **traza completa** del error (clase, método y línea).

⚙️ Variante 1

🔹 Observaciones

- No se crea un objeto `File dir` directamente.
- Se define la ruta base como **String**:

```
String ruta_V1 = "C:\\AD\\Ejercicio01\\Variante1";
File dir = new File(ruta_V1);
```

- `dir.mkdirs()` no lanza excepciones, por eso **no va dentro del try/catch**.
- Si el directorio se crea correctamente:

```
System.out.println("Directorio creado: " + dir.getAbsolutePath());
```

- Si no, mensaje de error y `return;` para finalizar ejecución.

🔹 Creación del fichero

```
File fichero = new File(dir, "fichero_de_texto.txt");
try {
    if (fichero.createNewFile()) {
        System.out.println("Fichero creado: " + fichero.getAbsolutePath());
    } else {
        System.out.println("El fichero ya existía: " + fichero.getAbsolutePath());
    }
} catch (IOException e) {
    System.out.println("ERROR al crear el fichero: " + e.getMessage());
}
```

- `createNewFile()` **sí lanza IOException**, por eso va en un **try/catch**.

⚙️ Variante 2

🔹 Estructura general

Esta variante incluye un **menú interactivo** con opciones:

```
1) Crear directorio
2) Crear fichero
3) Eliminar fichero
4) Eliminar directorio
5) Salir
```

🔹 Scanner y control de flujo

- Se utiliza `Scanner` para leer del teclado:

```
Scanner sc = new Scanner(System.in);
String opcion = sc.nextLine().trim();
```

- `.trim()` elimina espacios en blanco.
- Se usa **String en lugar de int** para poder aplicar `.trim()` (a `int` no se puede).

🔹 Estructura del switch-case

```
switch (opcion) {
    case "1": // Crear directorio
    case "2": // Crear fichero
    case "3": // Eliminar fichero
    case "4": // Eliminar directorio
    case "5": // Salir
    default:  // Opción incorrecta
}
```

🧱 Case 1 — Crear directorio

```
if (dirNuevo.exists()) {
    System.out.println("El directorio ya existe: " + dirNuevo.getAbsolutePath());
} else if (dirNuevo.mkdirs()) {
    System.out.println("Directorio creado: " + dirNuevo.getAbsolutePath());
} else {
    System.out.println("ERROR: No se pudo crear el directorio.");
}
```

📄 Case 2 — Crear fichero

```
if (!dirNuevo.exists()) {
    System.out.println("ERROR: No existe 'nuevoDirectorio'");
    break;
}
try {
    if (fichero2.createNewFile()) {
        System.out.println("Fichero creado: " + fichero2.getAbsolutePath());
    } else {
```

```
        System.out.println("El fichero ya existía: " + fichero2.getAbsolutePath());
    }
} catch (IOException e) {
    System.out.println("ERROR al crear el fichero: " + e.getMessage());
}
```

- Si no existe el directorio, se sale del `case` con **`break;`**.
- `createNewFile()` va **dentro del try/catch**.

🧹 Case 3 — Eliminar fichero

```
if (!fichero1.exists()) {
    System.out.println("No existe el fichero: " + fichero1.getAbsolutePath());
} else if (fichero1.delete()) {
    System.out.println("Fichero eliminado: " + fichero1.getAbsolutePath());
} else {
    System.out.println("ERROR: No se pudo eliminar el fichero.");
}
```

- `delete()` **no lanza excepciones** → no se usa try/catch.
- Si devuelve `true`, se eliminó correctamente.

🧱 Case 4 — Eliminar directorio

```
if (!dirNuevo.exists()) {
    System.out.println("No existe el directorio: " + dirNuevo.getAbsolutePath());
} else if (dirNuevo.delete()) {
    System.out.println("Directorio eliminado: " + dirNuevo.getAbsolutePath());
} else {
    System.out.println("ERROR: No se pudo eliminar el directorio (puede tener archivos
dentro).");
}
```

- Un directorio **no puede eliminarse si contiene ficheros**.
- Se muestra el mensaje de error predefinido.

🚪 Case 5 — Salir

```
salir = true;
System.out.println("Saliendo...");
```

⚠️ Default

```
System.out.println("Opción no válida. Intenta de nuevo.");
```

🧭 Referencia de métodos

Método	Descripción	Lanza excepción
`mkdir()`	Crea un único directorio	❌
`mkdirs()`	Crea todos los niveles necesarios	❌
`createNewFile()`	Crea un fichero vacío	✅ `IOException`
`delete()`	Elimina fichero/directorio vacío	❌
`exists()`	Comprueba si existe	❌
`getAbsolutePath()`	Devuelve la ruta completa	❌
`e.getMessage()`	Muestra mensaje corto del error	—
`e.printStackTrace()`	Muestra traza completa de error	—

⚙️ Tarea 2 — Ejercicio 1: Variante 3

Apuntes claros para la **Tarea 2 — Ejercicio 1 (Variante 3)**.

Ruta base: `C:\AD\Ejercicio1\Variante2`
Menú propuesto:

1. Crear directorio `nuevoDirectorio`
2. Crear fichero `fichero_de_texto2.txt` en `nuevoDirectorio`
3. Eliminar `C:\AD\Ejercicio1\Variante1\fichero_de_texto.txt`
4. Eliminar `nuevoDirectorio` **aunque tenga contenido**
5. Escribir **provincias de Andalucía** en `fichero_de_texto2.txt`
6. Salir

> Las **opciones 1–3 son idénticas a Variante2**. Mantén los mismos mensajes y validaciones.

🧱 Estructura del programa

- Constantes `File`:
  - `BASE`, `DIR_NUEVO`, `FICHERO2`, `FICHERO1_ANT`.
- Bucle con `Scanner` y `switch(opcion.trim())`.
- Mensajes por consola tras cada acción.

Para que compile correctamente necesitas importar las clases de **entrada/salida (java.io)** y las de **utilidades (java.util)**.

Tu archivo debería comenzar así:

```
package tarea.obligatoria.pkg1.clase.file.java;

import java.io.*;       // File, FileWriter, BufferedWriter, IOException
import java.util.*;     // Scanner, List, Arrays

public class TAREAOBLIGATORIA2Variante3 {
    // ... código aquí ...
}
```

Explicación breve:

- `java.io.*` importa todo lo necesario para trabajar con **archivos y flujos** (`File`, `FileWriter`, `BufferedWriter`, `IOException`).
- `java.util.*` incluye **Scanner**, **List**, **Arrays** y **Collections**, usados en el menú y en la lista de provincias.

---

1) Crear directorio

- `DIR_NUEVO.exists()` → informa si ya está.
- `DIR_NUEVO.mkdirs()` → crea todos los niveles.
- No lanza excepción.

```
if (DIR_NUEVO.exists()) { ... }
else if (DIR_NUEVO.mkdirs()) { ... }
else { System.out.println("ERROR: No se pudo crear el directorio."); }
```

---

2) Crear fichero en el directorio

- Requiere que exista `DIR_NUEVO`.
- `createNewFile()` devuelve `boolean` y **lanza** `IOException`.

```
if (!DIR_NUEVO.exists()) { System.out.println("ERROR: No existe 'nuevoDirectorio'");
return; }
try {
    if (FICHERO2.createNewFile()) { ... } else { ... }
} catch (IOException e) { System.out.println("ERROR al crear: " + e.getMessage()); }
```

---

3) Eliminar fichero del ejercicio anterior

- `FICHERO1_ANT.exists()` y `FICHERO1_ANT.delete()` → devuelve `boolean`.
- No hay `try/catch` porque **no lanza excepción**.

```
if (!FICHERO1_ANT.exists()) { ... }
else if (FICHERO1_ANT.delete()) { ... }
else { System.out.println("ERROR: No se pudo eliminar el fichero."); }
```

---

4) Eliminar `nuevoDirectorio` con contenido

Se usa una **función** para vaciar recursivamente y luego borrar el directorio. El `case` queda corto y legible.

En el `case "4"` :

```
if (!DIR_NUEVO.exists()) {
    System.out.println("No existe el directorio: " + DIR_NUEVO.getAbsolutePath());
} else {
    borrarContenido(DIR_NUEVO);          // función al final del archivo
    if (DIR_NUEVO.delete()) {
        System.out.println("Directorio eliminado: " + DIR_NUEVO.getAbsolutePath());
    } else {
```

```
        System.out.println("ERROR: No se pudo eliminar el directorio.");
    }
}
```

Función al final del código:

```
private static void borrarContenido(File dir) {
    File[] archivos = dir.listFiles();
    if (archivos != null) {
        for (File a : archivos) {
            if (a.isDirectory()) borrarContenido(a);  // recursión
            if (a.delete()) {
                System.out.println("Eliminado: " + a.getAbsolutePath());
            } else {
                System.out.println("No se pudo eliminar: " + a.getAbsolutePath());
            }
        }
    }
}
```

Qué hace:

- Lista hijos con `listFiles()`.
- Si es carpeta, entra recursivamente.
- Intenta `delete()` en cada hijo y reporta.
- Deja **vacío** `dir` para que su `delete()` funcione.

---

5) Escribir provincias de Andalucía en el fichero

- Provincias: Almería, Cádiz, Córdoba, Granada, Huelva, Jaén, Málaga, Sevilla.
- Usa **`BufferedWriter` envolviendo `FileWriter`**. Motivo: escribe en **bloques** con **buffer**, es **más eficiente** y `newLine()` añade saltos de línea portables. `FileWriter` escribe “al golpe” en cada `write()` y es menos eficiente con muchas líneas.

```
List<String> provincias = Arrays.asList(
    "Almería","Cádiz","Córdoba","Granada","Huelva","Jaén","Málaga","Sevilla"
);

// Garantiza que exista el directorio y el fichero destino
if (!DIR_NUEVO.exists()) DIR_NUEVO.mkdirs();

try (BufferedWriter bw = new BufferedWriter(new FileWriter(FICHERO2, false))) {
    for (String p : provincias) {
        bw.write(p);
        bw.newLine();                 // salto de línea independiente del SO
    }
    System.out.println("Provincias escritas en: " + FICHERO2.getAbsolutePath());
} catch (IOException e) {
    System.out.println("ERROR escribiendo: " + e.getMessage());
}
```

Notas rápidas sobre I/O:

- `FileWriter` / `FileReader` → simples, carácter a carácter. Más lentos con muchas operaciones.
- `BufferedWriter` / `BufferedReader` → envuelven con **buffer**. Más rápidos y cómodos (`newLine()` , lectura línea a línea).
- Usa **try-with-resources** para cerrar automáticamente.

🧭 Referencia de métodos

Método	Para qué	Lanza excepción
`exists()`	Comprobar presencia	No
`mkdirs()`	Crear árbol de directorios	No
`createNewFile()`	Crear fichero vacío	**Sí, `IOException`**
`delete()`	Borrar fichero o dir vacío	No
`isDirectory()`	Detectar carpeta	No
`listFiles()`	Obtener hijos de un dir	No
`getAbsolutePath()`	Ruta completa	No
`BufferedWriter.newLine()`	Salto de línea portable	No

Resolución Tarea 2

Ejercicio 1

```
package tarea.obligatoria.pkg1.clase.file.java;

import java.io.*;      // Clases necesarias para manejo de archivos (File, FileWriter,
IOException)
import java.util.*;    // Clases para listas, arrays y lectura de teclado (Scanner,
Arrays, List)

/**
 * Programa: TAREAOBLIGATORIA2Variante3
 * Asignatura: Acceso a Datos
 * Descripción: Gestión de archivos y directorios mediante menú de opciones.
 *              Incluye creación, eliminación y escritura en ficheros de texto.
 * Autor: David Rodríguez Igual
 * Fecha: 23 Octubre 2025
 */
public class TAREAOBLIGATORIA2Variante3 {

    // ===== RUTAS PRINCIPALES =====
    // Directorio base de trabajo
    private static final File BASE = new File("C:\\AD\\Ejercicio1\\Variante2");

    // Subdirectorio donde se crearán los archivos
    private static final File DIR_NUEVO = new File(BASE, "nuevoDirectorio");

    // Fichero nuevo dentro del directorio anterior
    private static final File FICHERO2 = new File(DIR_NUEVO, "fichero_de_texto2.txt");

    // Fichero del ejercicio anterior (Variante1) que se eliminará
    private static final File FICHERO1_ANT = new
```

```
File("C:\\AD\\Ejercicio1\\Variante1\\fichero_de_texto.txt");


    // ===== MÉTODO PRINCIPAL =====
    public static void main(String[] args) {
        // Crea la ruta base si no existe
        BASE.mkdirs();

        // Se usa Scanner para leer las opciones del usuario por consola
        try (Scanner sc = new Scanner(System.in)) {
            boolean salir = false;
            // Bucle principal del menú: se repite hasta elegir "Salir"
            while (!salir) {
                mostrarMenu();  // Muestra el menú en pantalla
                String op = sc.nextLine().trim(); // Lee opción sin espacios
                switch (op) {
                    case "1": crearDirectorio(); break;
                    case "2": crearFichero2(); break;
                    case "3": eliminarFicheroAnterior(); break;
                    case "4": eliminarDirectorioRecursivo(); break;
                    case "5": escribirProvinciasAndalucia(); break;
                    case "6": salir = true; System.out.println("Saliendo..."); break;
                    default: System.out.println("Opción no válida. Intenta de
nuevo.");
                }
            }
        }
    }


    // ===== MENÚ PRINCIPAL =====
    /**
     * Muestra las distintas opciones del programa al usuario.
     * Se ejecuta en cada iteración del bucle principal.
     */
    private static void mostrarMenu() {
        System.out.println("\n=== MENÚ ACCESO A DATOS ===");
        System.out.println("Ruta base: " + BASE.getAbsolutePath());
        System.out.println("1) Crear directorio 'nuevoDirectorio'");
        System.out.println("2) Crear fichero 'fichero_de_texto2.txt' en
'nuevoDirectorio'");
        System.out.println("3) Eliminar fichero del ejercicio anterior
'fichero_de_texto.txt'");
        System.out.println("4) Eliminar directorio 'nuevoDirectorio' (recursivo)");
        System.out.println("5) Escribir provincias de Andalucía en
'fichero_de_texto2.txt'");
        System.out.println("6) Salir");
        System.out.print("Elige opción: ");
    }


    // ===== OPCIÓN 1: CREAR DIRECTORIO =====
    /**
     * Crea el directorio nuevoDirectorio dentro de la ruta base.
```

```
     * Si ya existe, informa al usuario.
     */
    private static void crearDirectorio() {
        if (DIR_NUEVO.exists()) {
            System.out.println("El directorio ya existe: " +
DIR_NUEVO.getAbsolutePath());
        } else if (DIR_NUEVO.mkdirs()) {
            System.out.println("Directorio creado: " + DIR_NUEVO.getAbsolutePath());
        } else {
            System.out.println("ERROR: No se pudo crear el directorio.");
        }
    }


    // ===== OPCIÓN 2: CREAR FICHERO =====
    /**
     * Crea un fichero de texto dentro del directorio nuevoDirectorio.
     * Usa createNewFile(), que devuelve true si se crea correctamente.
     * Puede lanzar IOException si ocurre un error de entrada/salida.
     */
    private static void crearFichero2() {
        if (!DIR_NUEVO.exists()) {
            System.out.println("ERROR: No existe 'nuevoDirectorio'.");
            return;
        }
        try {
            if (FICHERO2.createNewFile()) {
                System.out.println("Fichero creado: " + FICHERO2.getAbsolutePath());
            } else {
                System.out.println("El fichero ya existía: " +
FICHERO2.getAbsolutePath());
            }
        } catch (IOException e) {
            System.out.println("ERROR al crear el fichero: " + e.getMessage());
        }
    }


    // ===== OPCIÓN 3: ELIMINAR FICHERO DEL EJERCICIO ANTERIOR =====
    /**
     * Elimina el fichero del ejercicio anterior (Variante1).
     * Usa delete(), que devuelve true si se borra correctamente.
     */
    private static void eliminarFicheroAnterior() {
        if (!FICHERO1_ANT.exists()) {
            System.out.println("No existe el fichero: " +
FICHERO1_ANT.getAbsolutePath());
        } else if (FICHERO1_ANT.delete()) {
            System.out.println("Fichero eliminado: " +
FICHERO1_ANT.getAbsolutePath());
        } else {
            System.out.println("ERROR: No se pudo eliminar el fichero.");
        }
    }
```

```java
    // ===== OPCIÓN 4: ELIMINAR DIRECTORIO RECURSIVO =====
    /**
     * Elimina el directorio nuevoDirectorio y todo su contenido.
     * Usa una función auxiliar recursiva llamada borrarContenido().
     */
    private static void eliminarDirectorioRecursivo() {
        if (!DIR_NUEVO.exists()) {
            System.out.println("No existe el directorio: " +
DIR_NUEVO.getAbsolutePath());
            return;
        }
        borrarContenido(DIR_NUEVO);
        if (DIR_NUEVO.delete()) {
            System.out.println("Directorio eliminado: " +
DIR_NUEVO.getAbsolutePath());
        } else {
            System.out.println("ERROR: No se pudo eliminar el directorio.");
        }
    }


    // ===== OPCIÓN 5: ESCRIBIR PROVINCIAS EN EL FICHERO =====
    /**
     * Escribe las provincias de Andalucía dentro del fichero fichero_de_texto2.txt.
     * Usa FileWriter para escribir línea por línea y flush() para limpiar el buffer.
     * Antes de escribir, comprueba que el fichero exista.
     */
    private static void escribirProvinciasAndalucia() {
        String[] provincias = {
                "Almería", "Cádiz", "Córdoba", "Granada",
                "Huelva", "Jaén", "Málaga", "Sevilla"
        };

        // Verificación de existencia del fichero antes de escribir
        if (!FICHERO2.exists()) {
            System.out.println("Error: No existe el fichero
'fichero_de_texto2.txt'.");
            return;
        }

        // Escritura de provincias usando FileWriter
        try (FileWriter fw = new FileWriter(FICHERO2)) {
            for (int i = 0; i < provincias.length; i++) {
                fw.write(provincias[i]);             // Escribe el nombre de la
provincia
                fw.write(System.lineSeparator());    // Inserta salto de línea
            }
            fw.flush(); // Limpia el buffer y asegura que se escriba todo
            System.out.println("Provincias escritas correctamente en el fichero.");
        } catch (IOException e) {
            System.out.println("ERROR al escribir en el fichero: " + e.getMessage());
        }
```

```
    }


    // ===== FUNCIÓN AUXILIAR PARA ELIMINACIÓN RECURSIVA =====
    /**
     * Recorre el contenido de un directorio y elimina todos los ficheros y
subdirectorios.
     * Si encuentra subcarpetas, las elimina primero (llamada recursiva).
     * @param dir Directorio a limpiar
     */
    private static void borrarContenido(File dir) {
        File[] hijos = dir.listFiles();  // Lista los archivos dentro del directorio
        if (hijos == null) return;       // Si está vacío o inaccesible, sale
        for (File h : hijos) {
            if (h.isDirectory()) borrarContenido(h); // Llamada recursiva para
subcarpetas
            if (h.delete()) {
                System.out.println("Eliminado: " + h.getAbsolutePath());
            } else {
                System.out.println("No se pudo eliminar: " + h.getAbsolutePath());
            }
        }
    }
}
```

Ejercicio 2

He mantenido la misma **estructura modular** y estilo de la *Variante 3*:

- Uso de un **paquete único** (`tarea.obligatoria.pkg1.clase.file.java`).
- Definición de rutas con **`File`** como constantes (`private static final File ...`).
- Comentarios tipo bloque en cada método (idéntico formato que en el Ejercicio 1).
- Aplicación del patrón de **métodos separados** (cada bloque de acción en su propio `private static void`).
- Control de errores con `try-with-resources` , `IOException` , y mensajes informativos.
- Reutilización de `System.lineSeparator()` para compatibilidad entre sistemas.
- Estructura del **flujo principal** **`main()`** sin menú, pero secuencial y con salida por consola.

En resumen, el **esqueleto**, la organización y la forma de comentar vienen íntegros del trabajo anterior.

🔹 Elementos nuevos creados

- Archivo destino `Empleados.txt` y su ruta.
- Array `EMPLEADOS` con 10 líneas tipo “ID;Nombre”.
- Cuatro métodos nuevos:
  1. `escribirConFileWriter()`
  2. `leerConFileReader()`
  3. `escribirConBufferedWriter()`
  4. `leerConBufferedReader()`
- Lectura y escritura doble: primero con las clases básicas (`FileWriter` , `FileReader`), luego con las **buffered**, que implementan la misma funcionalidad optimizada.

- Salidas por consola diferenciadas para cada bloque.

🔹 Nuevo contenido

El bloque de escritura/lectura **BufferedWriter / BufferedReader** está **basado directamente** en el material del PDF *“Diferencias entre usar FileReader_Writer y BufferedReader_Writer”*.
De ahí se aplicaron tres ideas concretas:

1. **FileWriter/FileReader** → lectura/escritura carácter a carácter (básico).
2. **BufferedWriter/BufferedReader** → uso de buffer en memoria para mayor eficiencia.
3. Inclusión de `newLine()` en lugar de `"\n"` y explicación de su portabilidad.

```
package tarea.obligatoria.pkg1.clase.file.java;

import java.io.*;  // Importa todas las clases necesarias para trabajar con archivos

/**
 * Programa: TAREAOBLIGATORIA2_Ejercicio2
 * Asignatura: Acceso a Datos
 * Descripción:
 * - Crea un fichero de texto llamado Empleados.txt en la ruta
C:\AD\Ejercicio1\Variante2
 * - Escribe 10 empleados (ID y nombre) dentro del fichero
 * - Luego lee su contenido de dos formas:
 *      1. Usando FileWriter y FileReader (métodos básicos)
 *      2. Usando BufferedWriter y BufferedReader (métodos optimizados)
 *
 * Objetivo didáctico:
 * Comprender las diferencias entre los flujos de entrada/salida directos (File)
 * y los flujos con buffer (Buffered), aplicando control de errores con IOException.
 *
 * Autor: David Rodríguez Igual
 * Fecha: 23 Octubre 2025
 */
public class TAREAOBLIGATORIA2_Ejercicio2 {

    // ===== RUTA DEL FICHERO =====
    // La clase File representa una ruta o un archivo físico en disco.
    // No crea el archivo por sí misma, solo "apunta" a él.
    private static final File FICHERO_EMPLEADOS =
            new File("C:\\AD\\Ejercicio1\\Variante2\\Empleados.txt");

    // ===== DATOS A ESCRIBIR =====
    // Array con 10 empleados, cada uno formado por un ID y un nombre.
    // Se usarán para escribir las líneas del fichero.
    private static final String[] EMPLEADOS = {
            "1;Ana Gómez", "2;Luis Pérez", "3;Marta Ruiz", "4;Iván Soto", "5;Noa
Vidal",
            "6;Sara León", "7;Hugo Mora", "8;Leo Navas", "9;Nora Gil", "10;Pau Roca"
    };

    // ===== MÉTODO PRINCIPAL =====
```

```java
    public static void main(String[] args) {
        // El método main() actúa como punto de entrada del programa.

        // 1. Escribimos y leemos el fichero con las clases básicas
FileWriter/FileReader.
        escribirConFileWriter();
        leerConFileReader();

        // 2. Luego repetimos con las clases optimizadas
BufferedWriter/BufferedReader.
        escribirConBufferedWriter();
        leerConBufferedReader();
    }

    // ===== 1️⃣ Escritura con FileWriter =====
    /**
     * FileWriter → clase de escritura de texto carácter a carácter directamente en
disco.
     * No usa buffer intermedio, por lo que cada operación de write() accede al
sistema de archivos.
     *
     * Este método:
     * - Crea (o sobrescribe) el fichero Empleados.txt
     * - Escribe una línea por empleado
     * - Añade salto de línea con System.lineSeparator()
     */
    private static void escribirConFileWriter() {
        System.out.println("\n--- Escritura con FileWriter ---");

        // mkdirs() crea los directorios intermedios si no existen.
        // No crea el archivo, solo las carpetas necesarias.
        FICHERO_EMPLEADOS.getParentFile().mkdirs();

        // try-with-resources: cierra automáticamente el flujo al finalizar el bloque.
        try (FileWriter fw = new FileWriter(FICHERO_EMPLEADOS)) {
            // Bucle for-each que recorre cada elemento del array EMPLEADOS
            for (String emp : EMPLEADOS) {
                fw.write(emp);                    // Escribe el texto del empleado (ID
y nombre)
                fw.write(System.lineSeparator()); // Inserta salto de línea compatible
con Windows
            }

            fw.flush(); // Envía cualquier dato pendiente del buffer interno al
archivo
            System.out.println("Fichero creado correctamente en: " +
FICHERO_EMPLEADOS.getAbsolutePath());
        } catch (IOException e) {
            // IOException: error general de entrada/salida (permiso, ruta, espacio,
etc.)
            System.out.println("ERROR al escribir con FileWriter: " + e.getMessage());
        }
    }
```

```java
    // ===== 2️⃣ Lectura con FileReader =====
    /**
     * FileReader → clase de lectura de texto carácter a carácter directamente desde
disco.
     * No usa buffer, por lo que cada read() lee un único carácter del archivo.
     *
     * Este método:
     * - Abre el fichero Empleados.txt
     * - Lee carácter a carácter hasta llegar al final (-1)
     * - Muestra su contenido por consola
     */
    private static void leerConFileReader() {
        System.out.println("\n--- Lectura con FileReader ---");

        try (FileReader fr = new FileReader(FICHERO_EMPLEADOS)) {
            int c; // Variable para almacenar cada carácter leído (valor ASCII)
            while ((c = fr.read()) != -1) { // -1 indica fin de archivo (EOF)
                System.out.print((char) c); // Convierte el valor numérico en carácter
            }
        } catch (IOException e) {
            System.out.println("ERROR al leer con FileReader: " + e.getMessage());
        }
    }

    // ===== 3️⃣ Escritura con BufferedWriter =====
    /**
     * BufferedWriter → clase de escritura con buffer intermedio en memoria.
     *
     * Diferencias con FileWriter:
     * - Almacena temporalmente los datos antes de volcarlos al disco, reduciendo
accesos físicos.
     * - Usa el método newLine() en lugar de System.lineSeparator().
     *
     * Este método:
     * - Reescribe el fichero Empleados.txt (sobrescribe el contenido anterior)
     * - Escribe los mismos empleados de forma más eficiente
     */
    private static void escribirConBufferedWriter() {
        System.out.println("\n--- Escritura con BufferedWriter ---");

        // El BufferedWriter se asocia a un FileWriter interno
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter(FICHERO_EMPLEADOS))) {
            for (String emp : EMPLEADOS) {
                bw.write(emp);  // Escribe la línea con ID y nombre
                bw.newLine();   // Añade salto de línea de forma segura y portable
            }
            // flush() no es necesario aquí porque el try-with-resources ya cierra el
flujo correctamente
            System.out.println("Fichero reescrito correctamente con BufferedWriter.");
        } catch (IOException e) {
            System.out.println("ERROR al escribir con BufferedWriter: " +
e.getMessage());
        }
```

```
    }

    // ===== 4️⃣ Lectura con BufferedReader =====
    /**
     * BufferedReader → clase de lectura de texto con buffer intermedio en memoria.
     *
     * Diferencias con FileReader:
     * - Permite leer líneas completas usando readLine()
     * - Mayor rendimiento porque reduce las operaciones de disco.
     *
     * Este método:
     * - Lee el contenido del fichero Empleados.txt línea por línea
     * - Muestra cada línea en consola
     */
    private static void leerConBufferedReader() {
        System.out.println("\n--- Lectura con BufferedReader ---");

        try (BufferedReader br = new BufferedReader(new
FileReader(FICHERO_EMPLEADOS))) {
            String linea; // Variable para guardar cada línea leída
            while ((linea = br.readLine()) != null) { // readLine() devuelve null al
final del archivo
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.out.println("ERROR al leer con BufferedReader: " + e.getMessage());
        }
    }
}
```

🧭 Referencia de clases y métodos

Clase / Método	Tipo	Propósito
`File`	Clase	Representa un archivo o directorio. Puede apuntar a algo que todavía no existe.
`FileWriter`	Clase	Escribe texto carácter a carácter directamente en un archivo. Lenta pero simple.
`FileReader`	Clase	Lee texto carácter a carácter desde un archivo.
`BufferedWriter`	Clase	Capa superior de `FileWriter`. Usa un buffer para acumular texto en memoria y escribirlo de golpe.
`BufferedReader`	Clase	Capa superior de `FileReader`. Permite leer líneas completas con `readLine()`.
`try-with-resources`	Bloque	Cierra automáticamente los objetos que implementan `AutoCloseable`, como `FileWriter` o `BufferedReader`.
`flush()`	Método	Limpia el buffer forzando la escritura inmediata al disco.
`newLine()`	Método	Inserta un salto de línea dependiente del sistema (`\r\n` en Windows, `\n` en Linux).
`System.lineSeparator()`	Método estático	Alternativa genérica para añadir un salto de línea según el sistema operativo.